CHAPTER 2

Getting Started

The best way to appreciate a software development framework is to jump right in and use it. In this chapter, I explain how to prepare for ASP.NET Core development and how to create and run an ASP.NET Core application.

UPDATES TO THIS BOOK

Microsoft has an active development schedule for .NET and ASP.NET Core, which means that there may be new releases available by the time you read this book. It doesn't seem fair to expect readers to buy a new book every few months, especially since most changes are relatively minor. Instead, I will post free updates to the GitHub repository for this book (`https://github.com/apress/pro-asp.net-core-6`) for breaking changes.

This kind of update is an ongoing experiment for me (and for Apress), and it continues to evolve—not least because I don't know what the future major releases of ASP.NET Core will contain—but the goal is to extend the life of this book by supplementing the examples it contains.

I am not making any promises about what the updates will be like, what form they will take, or how long I will produce them before folding them into a new edition of this book. Please keep an open mind and check the repository for this book when new ASP.NET Core versions are released. If you have ideas about how the updates could be improved, then email me at `adam@adam-freeman.com` and let me know.

Choosing a Code Editor

Microsoft provides a choice of tools for ASP.NET Core development: Visual Studio and Visual Studio Code. Visual Studio is the traditional development environment for .NET applications, and it offers an enormous range of tools and features for developing all sorts of applications. But it can be resource-hungry and slow, and some of the features are so determined to be helpful they get in the way of development.

Visual Studio Code is a lightweight alternative that doesn't have the bells and whistles of Visual Studio but is perfectly capable of handling ASP.NET Core development.

All the examples in this book include instructions for both editors, and both Visual Studio and Visual Studio Code can be used without charge, so you can use whichever suits your development style.

A. Freeman, *Pro ASP.NET Core 6*, https://doi.org/10.1007/978-1-4842-7957-1_2

If you are new to .NET development, then start with Visual Studio. It provides more structured support for creating the different types of files used in ASP.NET Core development, which will help ensure you get the expected results from the code examples.

■ **Note** This book describes ASP.NET Core development for Windows. It is possible to develop and run ASP.NET Core applications on Linux and macOS, but most readers use Windows, and that is what I have chosen to focus on. Almost all the examples in this book rely on LocalDB, which is a Windows-only feature provided by SQL Server that is not available on other platforms. If you want to follow this book on another platform, then you can contact me using the email address in Chapter 1, and I will try to help you get started.

Installing Visual Studio

ASP.NET Core 6 requires Visual Studio 2022. I use the free Visual Studio 2022 Community Edition, which can be downloaded from `www.visualstudio.com`. Run the installer, and you will see the prompt shown in Figure 2-1.

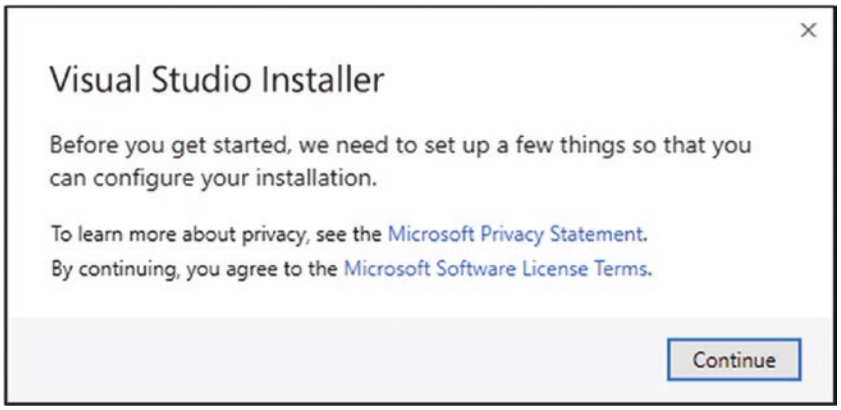

Figure 2-1. *Starting the Visual Studio installer*

Click the Continue button, and the installer will download the installation files, as shown in Figure 2-2.

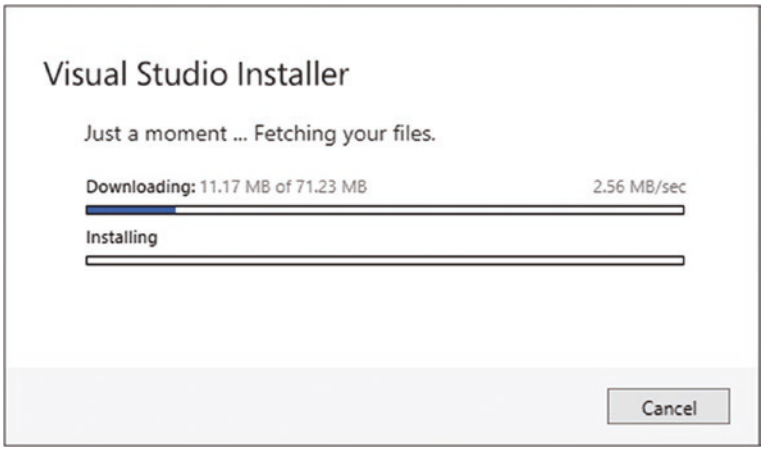

Figure 2-2. *Downloading the Visual Studio installer files*

When the installer files have been downloaded, you will be presented with a set of installation options, grouped into workloads. Ensure that the "ASP.NET and web development" workload is checked, as shown in Figure 2-3.

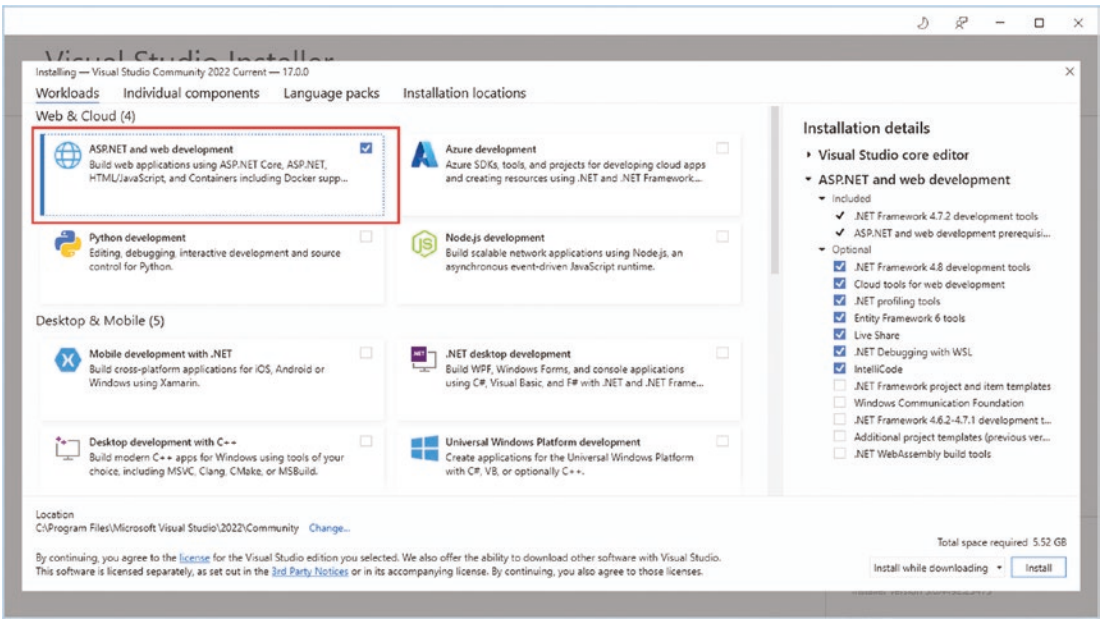

Figure 2-3. *Selecting the workload*

Select the "Individual components" section at the top of the window and ensure the SQL Server Express 2019 LocalDB option is checked, as shown in Figure 2-4. This is the database component that I will be using to store data in later chapters.

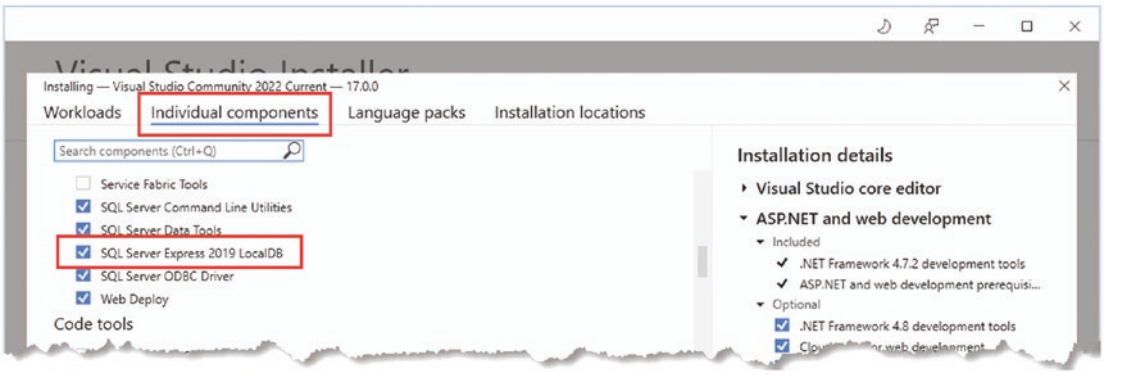

Figure 2-4. *Ensuring LocalDB is installed*

Click the Install button, and the files required for the selected workload will be downloaded and installed. To complete the installation, a reboot may be required.

Installing the .NET SDK

The Visual Studio installer will install the .NET Software Development Kit (SDK), but it may not install the version required for the examples in this book. Go to `https://dotnet.microsoft.com/download/dotnet-core/6.0` and download the installer for version 6.0.0 of the .NET SDK, which is the long-term support release at the time of writing. Run the installer; once the installation is complete, open a new PowerShell command prompt from the Windows Start menu and run the command shown in Listing 2-1, which displays a list of the installed .NET SDKs.

Listing 2-1. Listing the Installed SDKs

```
dotnet --list-sdks
```

Here is the output from a fresh installation on a Windows machine that has not been used for .NET:

```
6.0.100 [C:\Program Files\dotnet\sdk]
```

If you have been working with different versions of .NET, you may see a longer list, like this one:

```
3.1.101 [C:\Program Files\dotnet\sdk]
5.0.100 [C:\Program Files\dotnet\sdk]
5.0.401 [C:\Program Files\dotnet\sdk]
6.0.100 [C:\Program Files\dotnet\sdk]
```

Regardless of how many entries there are, you must ensure there is one for the 6.0.1xx version, where the last two digits may differ.

Installing Visual Studio Code

If you have chosen to use Visual Studio Code, download the installer from https://code.visualstudio.com. No specific version is required, and you should select the current stable build. Run the installer and ensure you check the Add to PATH option, as shown in Figure 2-5.

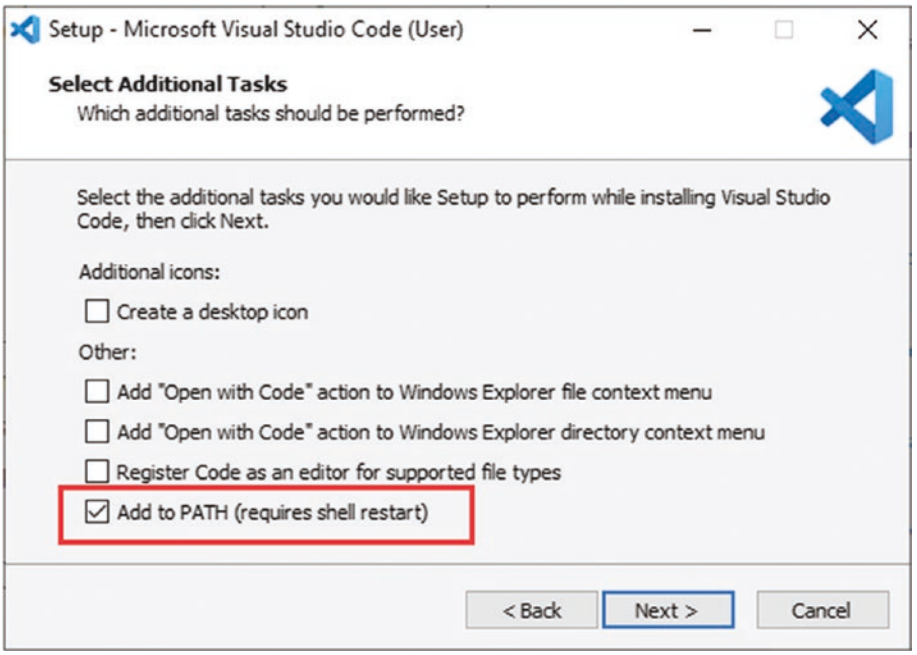

Figure 2-5. *Configuring the Visual Studio Code installation*

Installing the .NET SDK

The Visual Studio Code installer does not include the .NET SDK, which must be installed separately. Go to https://dotnet.microsoft.com/download/dotnet-core/6.0 and download the installer for version 6.0.0 of the .NET SDK, which is the long-term support release at the time of writing. Run the installer; once the installation is complete, open a new PowerShell command prompt from the Windows Start menu and run the command shown in Listing 2-2, which displays a list of the installed .NET SDKs.

Listing 2-2. Listing the Installed SDKs

```
dotnet --list-sdks
```

Here is the output from a fresh installation on a Windows machine that has not been used for .NET:

```
6.0.100 [C:\Program Files\dotnet\sdk]
```

If you have been working with different versions of .NET, you may see a longer list, like this one:

```
3.1.101 [C:\Program Files\dotnet\sdk]
5.0.100 [C:\Program Files\dotnet\sdk]
5.0.401 [C:\Program Files\dotnet\sdk]
6.0.100 [C:\Program Files\dotnet\sdk]
```

Regardless of how many entries there are, you must ensure there is one for the 6.0.1xx version, where the last two digits may differ.

Installing SQL Server LocalDB

The database examples in this book require LocalDB, which is a zero-configuration version of SQL Server that can be installed as part of the SQL Server Express edition, which is available for use without charge from https://www.microsoft.com/en-in/sql-server/sql-server-downloads. Download and run the Express edition installer and select the Custom option, as shown in Figure 2-6.

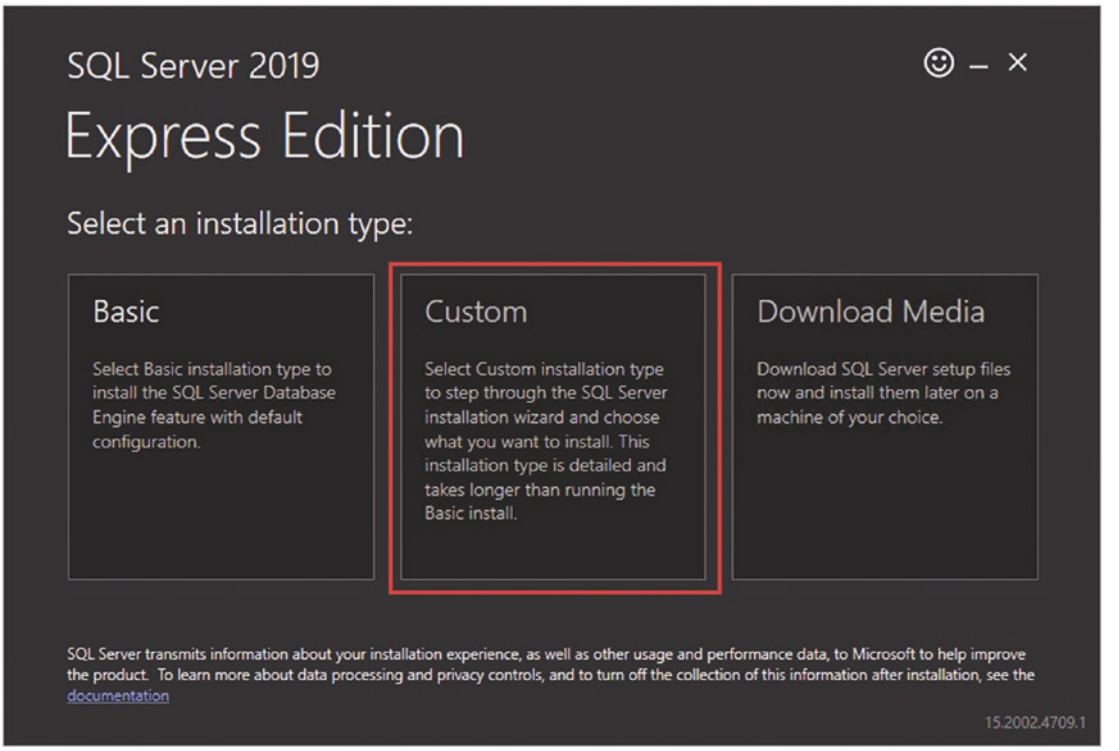

Figure 2-6. *Selecting the installation option for SQL Server*

Once you have selected the Custom option, you will be prompted to select a download location for the installation files. Click the Install button, and the download will begin.

When prompted, select the option to create a new SQL Server installation, as shown in Figure 2-7.

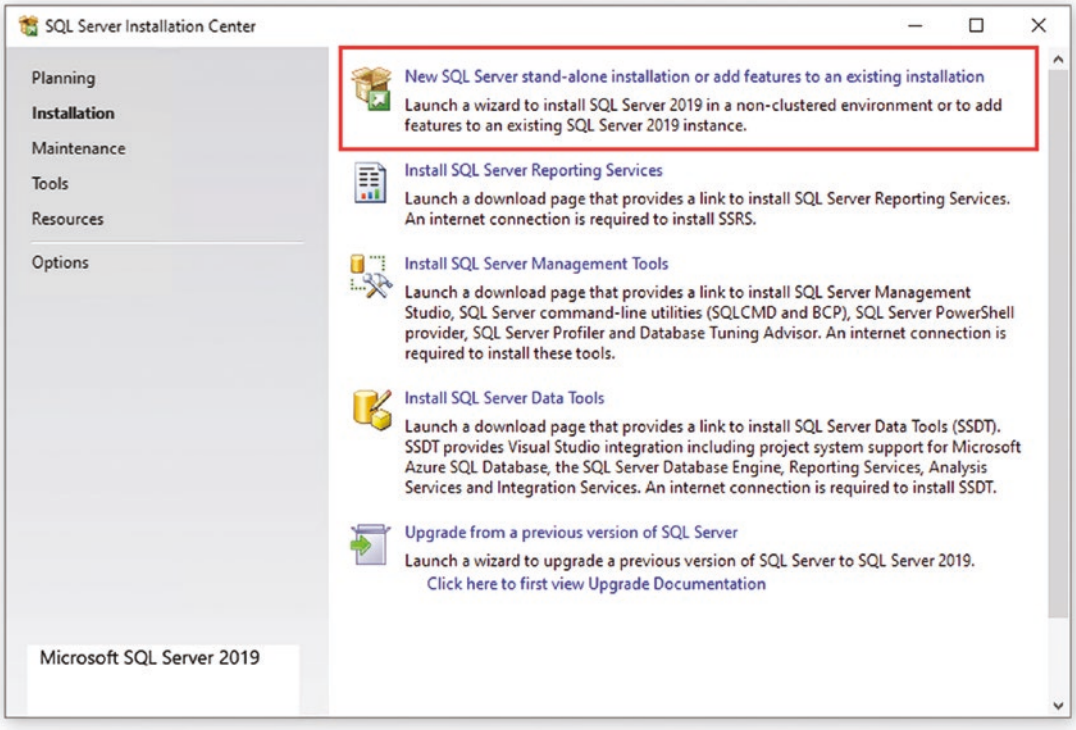

Figure 2-7. *Selecting an installation option*

Work through the installation process, selecting the default options as they are presented. When you reach the Feature Selection page, ensure that the LocalDB option is checked, as shown in Figure 2-8. (You may want to uncheck the options for R and Python, which are not used in this book and take a long time to download and install.)

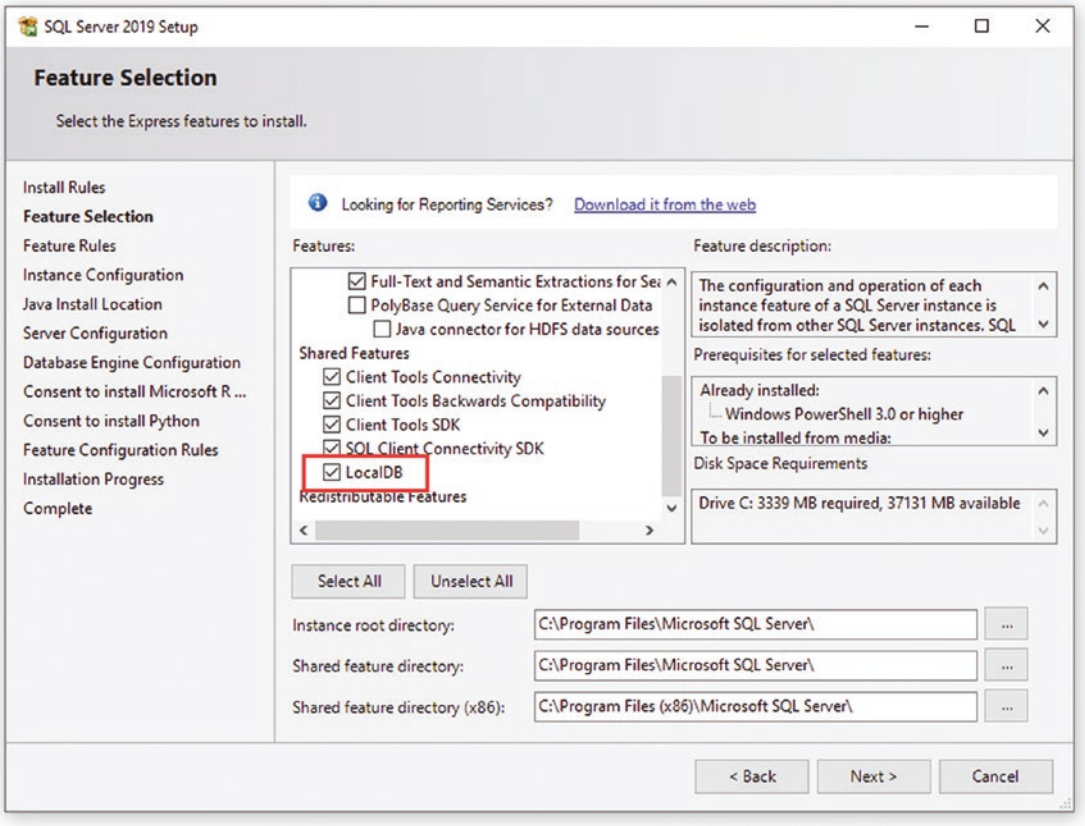

Figure 2-8. *Selecting the LocalDB feature*

On the Instance Configuration page, select the "Default instance" option, as shown in Figure 2-9.

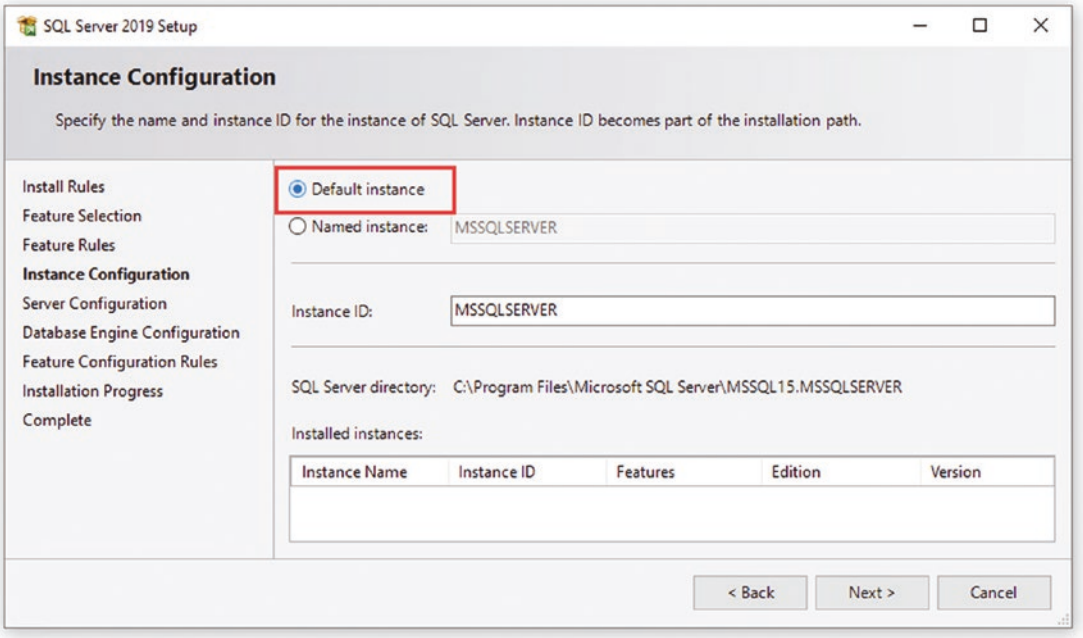

Figure 2-9. *Configuring the database*

Continue to work through the installation process, selecting the default values. Once the installation is complete, install the latest cumulative update for SQL Server. At the time of writing, the latest update is available at `https://support.microsoft.com/en-us/topic/kb5005679-cumulative-update-13-for-sql-server-2019-5c1be850-460a-4be4-a569-fe11f0adc535` but it is easier to reach this URL by searching for `KB5005679`. Newer updates may have been released by the time you read this chapter.

■ **Caution** It can be tempting to skip the update stage, but it is important to perform this step to get the expected results from the examples in this book.

Creating an ASP.NET Core Project

The most direct way to create a project is to use the command line. Open a new PowerShell command prompt from the Windows Start menu, navigate to the folder where you want to create your ASP.NET Core projects, and run the commands shown in Listing 2-3.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/apress/pro-asp.net-core-6`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 2-3. Creating a New Project

```
dotnet new globaljson --sdk-version 6.0.100 --output FirstProject
dotnet new mvc --no-https --output FirstProject --framework net6.0
dotnet new sln -o FirstProject
dotnet sln FirstProject add FirstProject
```

The first command creates a folder named `FirstProject` and adds to it a file named `global.json`, which specifies the version of .NET that the project will use; this ensures you get the expected results when following the examples. The second command creates a new ASP.NET Core project. The .NET SDK includes a range of templates for starting new projects, and the `mvc` template is one of the options available for ASP.NET Core applications. This project template creates a project that is configured for the MVC Framework, which is one of the application types supported by ASP.NET Core. Don't be intimidated by the idea of choosing a framework, and don't worry if you have not heard of MVC—by the end of the book, you will understand the features that each offers and how they fit together. The remaining commands create a solution file, which allows multiple projects to be used together.

■ **Note** This is one of a small number of chapters in which I use a project template that contains placeholder content. I don't like using predefined project templates because they encourage developers to treat important features, such as authentication, as black boxes. My goal in this book is to give you the knowledge to understand and manage every aspect of your ASP.NET Core applications, and that's why I start with an empty ASP.NET Core project. This chapter is about getting started quickly, for which the `mvc` template is well-suited.

Opening the Project Using Visual Studio

Start Visual Studio and click the "Open a project or solution" button, as shown in Figure 2-10.

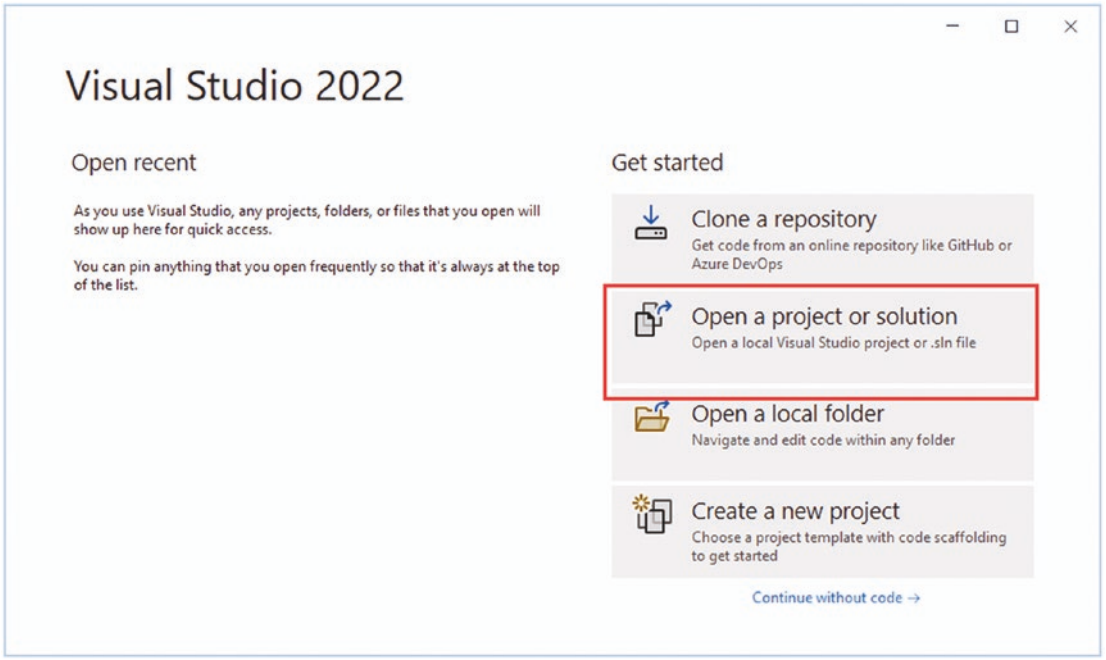

Figure 2-10. *Opening the ASP.NET Core project*

Navigate to the `FirstProject` folder, select the `FirstProject.sln` file, and click the Open button. Visual Studio will open the project and display its contents in the Solution Explorer window, as shown in Figure 2-11. The files in the project were created by the project template.

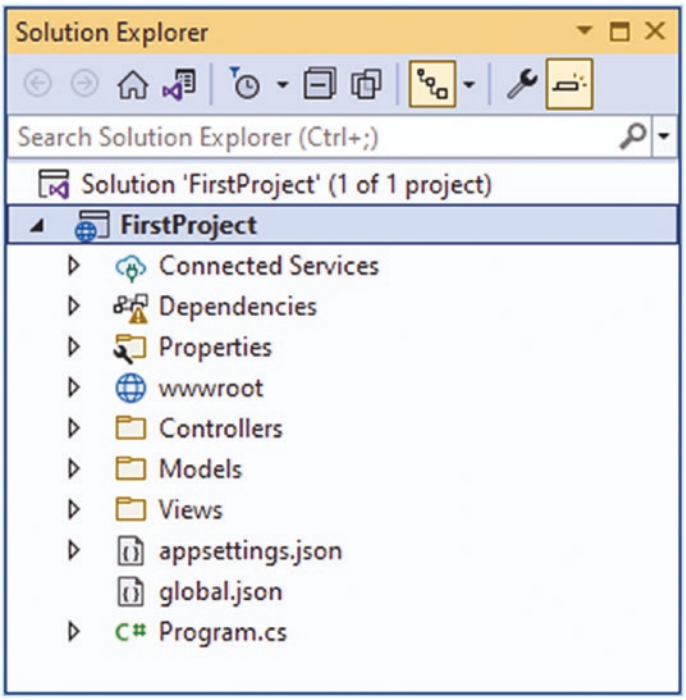

Figure 2-11. *Opening the project in Visual Studio*

Opening the Project with Visual Studio Code

Start Visual Studio Code and select File ➤ Open Folder. Navigate to the `FirstProject` folder and click the Select Folder button. Visual Studio Code will open the project and display its contents in the Explorer pane, as shown in Figure 2-12. (The default dark theme used in Visual Studio Code doesn't show well on the page, so I have changed to the light theme for the screenshots in this book.)

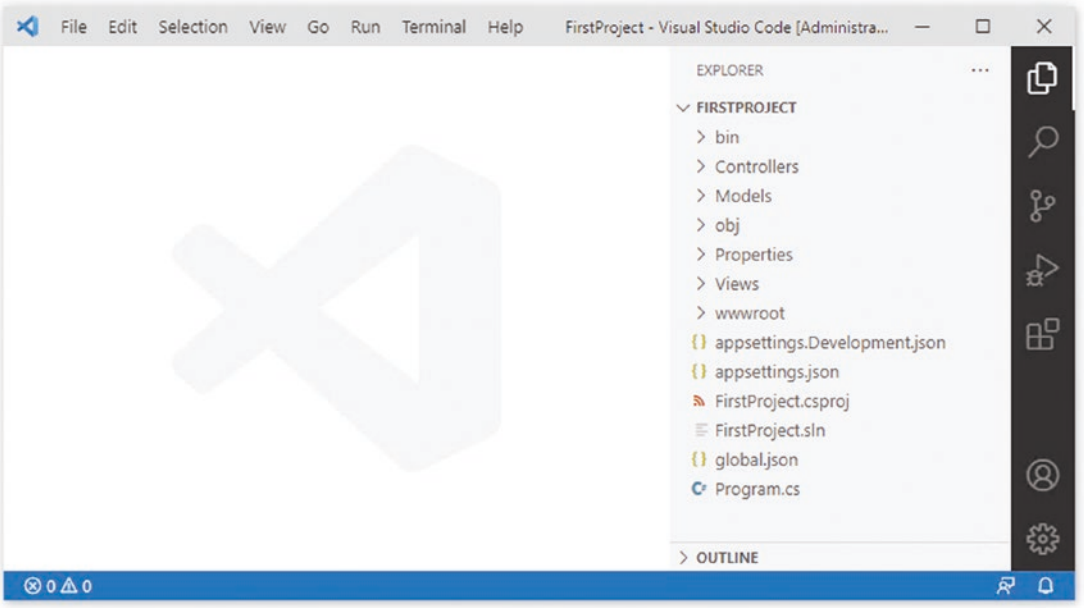

Figure 2-12. *Opening the project in Visual Studio Code*

Additional configuration is required the first time you open a .NET project in Visual Studio Code. The first step is to click the `Program.cs` file in the Explorer pane. This will trigger a prompt from Visual Studio Code to install the features required for C# development, as shown in Figure 2-13. If you have not opened a C# project before, you will see a prompt that offers to install the required assets, also shown in Figure 2-13.

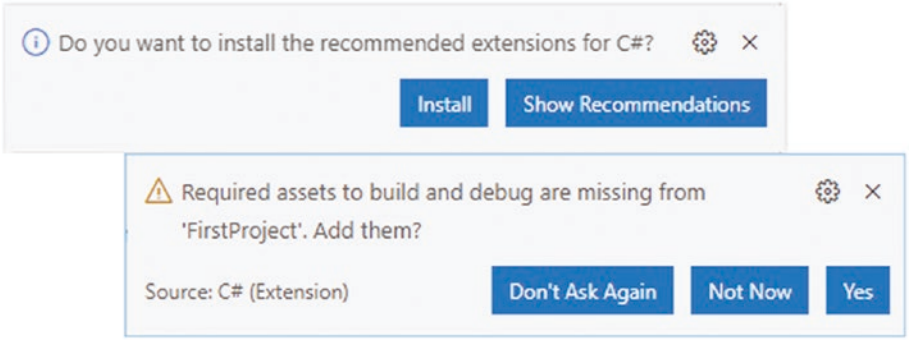

Figure 2-13. *Installing Visual Studio Code C# features*

Click the Install or Yes button, as appropriate, and Visual Studio Code will download and install the features required for .NET projects.

Running the ASP.NET Core Application

Visual Studio and Visual Studio Code can both run projects directly, but I use the command line tools throughout this book because they are more reliable and work more consistently, helping to ensure you get the expected results from the examples.

When the project is created, a file named `launchSettings.json` is created in the `Properties` folder, and it is this file that determines which HTTP port ASP.NET Core will use to listen for HTTP requests. Open this file in your chosen editor and change the ports in the URLs it contains to 5000, as shown in Listing 2-4.

Listing 2-4. Setting the HTTP Port in the launchSettings.json File in the Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "FirstProject": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

It is only the URL in the `profiles` section that affects the .NET command-line tools, but I have changed both of them to avoid any problems. Open a new PowerShell command prompt from the Windows Start menu; navigate to the `FirstProject` project folder, which is the folder that contains the `FirstProject.csproj` file; and run the command shown in Listing 2-5.

Listing 2-5. Starting the Example Application

```
dotnet run
```

The `dotnet run` command compiles and starts the project. Once the application has started, open a new browser window and request `http://localhost:5000`, which will produce the response shown in Figure 2-14.

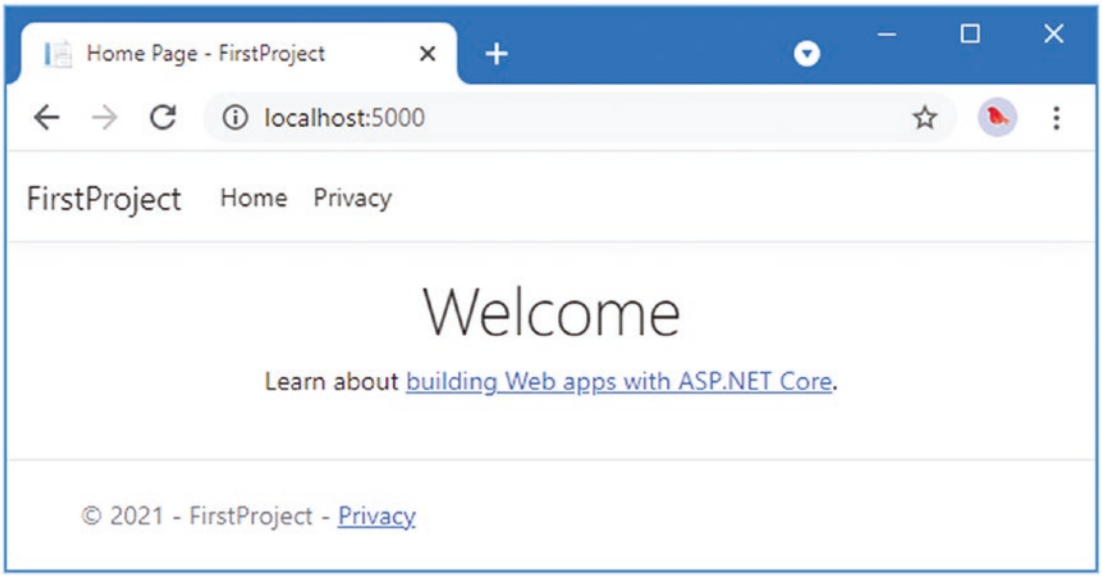

Figure 2-14. *Running the example project*

When you are finished, use Control+C to stop the ASP.NET Core application.

Understanding Endpoints

In an ASP.NET Core application, incoming requests are handled by *endpoints*. The endpoint that produced the response in Figure 2-14 is an *action*, which is a method that is written in C#. An action is defined in a *controller*, which is a C# class that is derived from the `Microsoft.AspNetCore.Mvc.Controller` class, the built-in controller base class.

Each public method defined by a controller is an action, which means you can invoke the action method to handle an HTTP request. The convention in ASP.NET Core projects is to put controller classes in a folder named `Controllers`, which was created by the template used to set up the project.

The project template added a controller to the `Controllers` folder to help jump-start development. The controller is defined in the class file named `HomeController.cs`. Controller classes contain a name followed by the word `Controller`, which means that when you see a file called `HomeController.cs`, you know that it contains a controller called `Home`, which is the default controller that is used in ASP.NET Core applications.

■ **Tip** Don't worry if the terms *controller* and *action* don't make immediate sense. Just keep following the example, and you will see how the HTTP request sent by the browser is handled by C# code.

Find the HomeController.cs file in the Solution Explorer or Explorer pane and click it to open it for editing. You will see the following code:

```
using System.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using FirstProject.Models;

namespace FirstProject.Controllers;

public class HomeController : Controller {
    private readonly ILogger<HomeController> _logger;

    public HomeController(ILogger<HomeController> logger) {
        _logger = logger;
    }

    public IActionResult Index() {
        return View();
    }

    public IActionResult Privacy() {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
         NoStore = true)]
    public IActionResult Error() {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id
            ?? HttpContext.TraceIdentifier });
    }
}
```

Using the code editor, replace the contents of the HomeController.cs file so that it matches Listing 2-6. I have removed all but one of the methods, changed the result type and its implementation, and removed the using statements for unused namespaces.

Listing 2-6. Changing the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace FirstProject.Controllers {

    public class HomeController : Controller {

        public string Index() {
            return "Hello World";
        }
    }
}
```

The result is that the Home controller defines a single action, named Index. These changes don't produce a dramatic effect, but they make for a nice demonstration. I have changed the method named Index so that it returns the string Hello World. Using the PowerShell prompt, run the dotnet run command in the FirstProject folder again and use the browser to request http://localhost:5000. The configuration of the project created by the template in Listing 2-6 means the HTTP request will be processed by the Index action defined by the Home controller. Put another way, the request will be processed by the Index method defined by the HomeController class. The string produced by the Index method is used as the response to the browser's HTTP request, as shown in Figure 2-15.

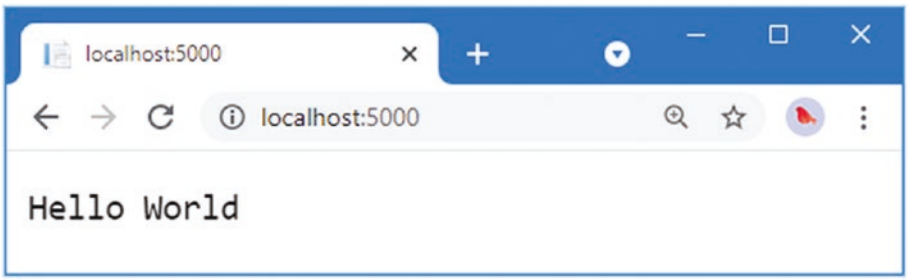

Figure 2-15. *The output from the action method*

Understanding Routes

The ASP.NET Core *routing* system is responsible for selecting the endpoint that will handle an HTTP request. A *route* is a rule that is used to decide how a request is handled. When the project was created, a default rule was created to get started. You can request any of the following URLs, and they will be dispatched to the Index action defined by the Home controller:

```
/
/Home
/Home/Index
```

So, when a browser requests http://yoursite/ or http://yoursite/Home, it gets back the output from HomeController's Index method. You can try this yourself by changing the URL in the browser. At the moment, it will be http://localhost:5000/, except that the port part may be different if you are using Visual Studio. If you append /Home or /Home/Index to the URL and press Return, you will see the same Hello World result from the application.

Understanding HTML Rendering

The output from the previous example wasn't HTML—it was just the string Hello World. To produce an HTML response to a browser request, I need a *view*, which tells ASP.NET Core how to process the result produced by the Index method into an HTML response that can be sent to the browser.

Creating and Rendering a View

The first thing I need to do is modify my Index action method, as shown in Listing 2-7. The changes are shown in bold, which is a convention I follow throughout this book to make the examples easier to follow.

Listing 2-7. Rendering a View in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace FirstProject.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            return View("MyView");
        }
    }
}
```

When I return a `ViewResult` object from an action method, I am instructing ASP.NET Core to *render* a view. I create the `ViewResult` by calling the `View` method, specifying the name of the view that I want to use, which is `MyView`.

Use Control+C to stop ASP.NET Core and then use the `dotnet run` command to compile and start it again. Use the browser to request `http://localhost:5000`, and you will see ASP.NET Core trying to find the view, as shown by the error message displayed in Figure 2-16.

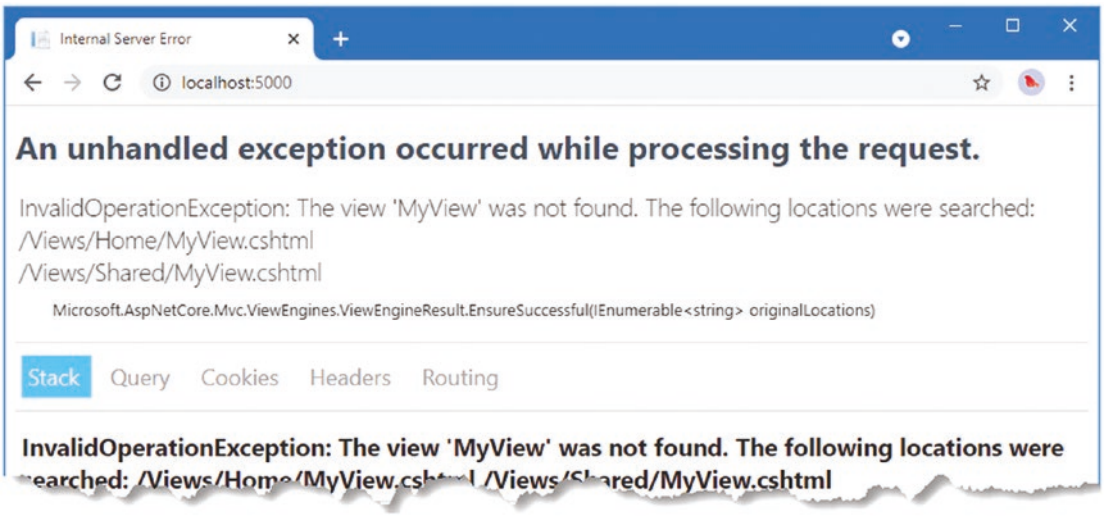

Figure 2-16. *Trying to find a view*

This is a helpful error message. It explains that ASP.NET Core could not find the view I specified for the action method and explains where it looked. Views are stored in the `Views` folder, organized into subfolders. Views that are associated with the `Home` controller, for example, are stored in a folder called `Views/Home`. Views that are not specific to a single controller are stored in a folder called `Views/Shared`. The template used to create the project added the `Home` and `Shared` folders automatically and added some placeholder views to get the project started.

If you are using Visual Studio, right-click the `Views/Home` folder in the Solution Explorer and select Add ➤ New Item from the pop-up menu. Visual Studio will present you with a list of templates for adding items to the project. Locate the Razor View - Empty item, which can be found in the ASP.NET Core ➤ Web ➤ ASP.NET section, as shown in Figure 2-17. Set the name of the new file to `MyView.cshtml` and click the Add button. Visual Studio will add a file named `MyView.cshtml` to the `Views/Home` folder and will open it for editing. Replace the contents of the file with those shown in Listing 2-8.

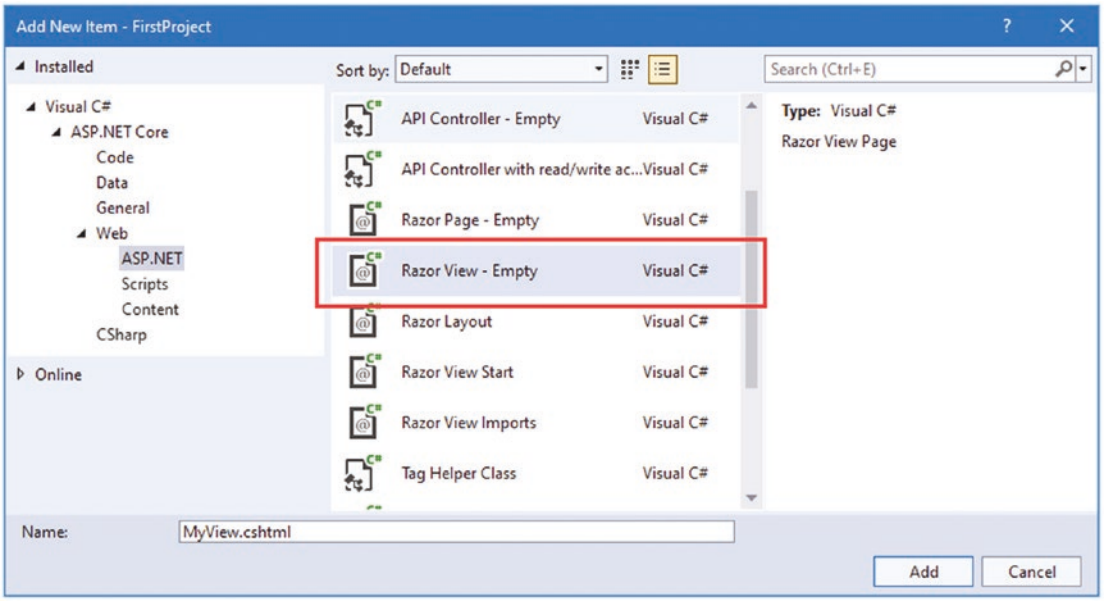

Figure 2-17. *Selecting a Visual Studio item template*

Visual Studio Code doesn't provide item templates. Instead, right-click the `Views/Home` folder in the file explorer pane and select New File from the pop-up menu. Set the name of the file to `MyView.cshtml` and press Return. The file will be created and opened for editing. Add the content shown in Listing 2-8.

■ **Tip** It is easy to end up creating the view file in the wrong folder. If you didn't end up with a file called `MyView.cshtml` in the `Views/Home` folder, then either drag the file into the correct folder or delete the file and try again.

Listing 2-8. The Contents of the MyView.cshtml File in the Views/Home Folder

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Hello World (from the view)
    </div>
</body>
</html>
```

The new contents of the view file are mostly HTML. The exception is the part that looks like this:

```
...
@{
    Layout = null;
}
...
```

This is an expression that will be interpreted by Razor, which is the component that processes the contents of views and generates HTML that is sent to the browser. Razor is a *view engine*, and the expressions in views are known as *Razor expressions*.

The Razor expression in Listing 2-8 tells Razor that I chose not to use a layout, which is like a template for the HTML that will be sent to the browser (and which I describe in Chapter 22). To see the effect of creating the view, use Control+C to stop ASP.NET Core if it is running and use the `dotnet run` command to compile and start the application again. Use a browser to request `http://localhost:5000`, and you will see the result shown in Figure 2-18.

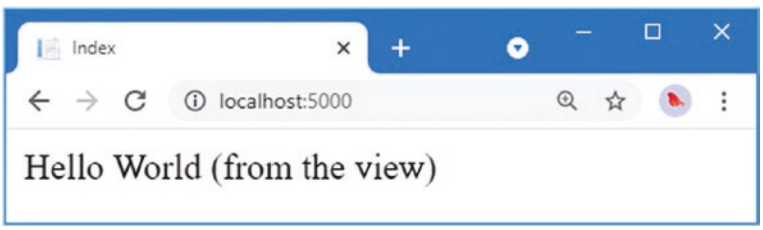

Figure 2-18. *Rendering a view*

When I first edited the `Index` action method, it returned a `string` value. This meant that ASP.NET Core did nothing except pass the string value as is to the browser. Now that the `Index` method returns a `ViewResult`, Razor is used to process a view and render an HTML response. Razor was able to locate the

view because I followed the standard naming convention, which is to put view files in a folder whose name matched the controller that contains the action method. In this case, this meant putting the view file in the Views/Home folder, since the action method is defined by the Home controller.

I can return other results from action methods besides strings and ViewResult objects. For example, if I return a RedirectResult, the browser will be redirected to another URL. If I return an HttpUnauthorizedResult, I can prompt the user to log in. These objects are collectively known as *action results*. The action result system lets you encapsulate and reuse common responses in actions. I'll tell you more about them and explain the different ways they can be used in Chapter 19.

Adding Dynamic Output

The whole point of a web application is to construct and display *dynamic* output. The job of the action method is to construct data and pass it to the view so it can be used to create HTML content based on the data values. Action methods provide data to views by passing arguments to the View method, as shown in Listing 2-9. The data provided to the view is known as the *view model*.

Listing 2-9. Providing a View Model in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace FirstProject.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            string viewModel = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View("MyView", viewModel);
        }
    }
}
```

The view model in this example is a string, and it is provided to the view as the second argument to the View method. Listing 2-10 updates the view so that it receives and uses the view model in the HTML it generates.

Listing 2-10. Using a View Model in the MyView.cshtml File in the Views/Home Folder

```
@model string
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
```

```
<body>
    <div>
        @Model World (from the view)
    </div>
</body>
</html>
```

The type of the view model is specified using the `@model` expression, with a lowercase *m*. The view model value is included in the HTML output using the `@Model` expression, with an uppercase *M*. (It can be difficult at first to remember which is lowercase and which is uppercase, but it soon becomes second nature.)

When the view is rendered, the view model data provided by the action method is inserted into the HTML response. Use Control+C to stop ASP.NET Core and use the `dotnet run` command to build and start it again. Use a browser to request `http://localhost:5000`, and you will see the output shown in Figure 2-19 (although you may see the morning greeting if you are following this example before midday).

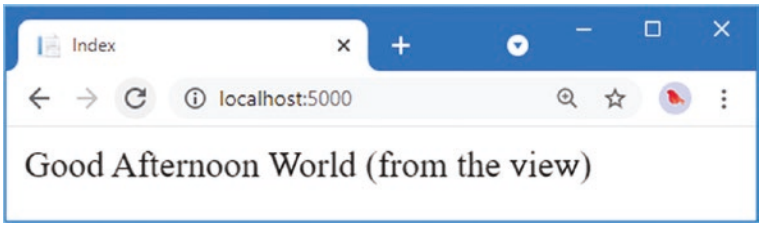

Figure 2-19. *Generating dynamic content*

Putting the Pieces Together

It is a simple result, but this example reveals all the building blocks you need to create a simple ASP.NET Core web application and to generate a dynamic response. The ASP.NET Core platform receives an HTTP request and uses the routing system to match the request URL to an endpoint. The endpoint, in this case, is the `Index` action method defined by the `Home` controller. The method is invoked and produces a `ViewResult` object that contains the name of a view and a view model object. The Razor view engine locates and processes the view, evaluating the `@Model` expression to insert the data provided by the action method into the response, which is returned to the browser and displayed to the user. There are, of course, many other features available, but this is the essence of ASP.NET Core, and it is worth bearing this simple sequence in mind as you read the rest of the book.

Summary

In this chapter, I explained how to get set up for ASP.NET Core development by installing Visual Studio or Visual Studio Code and the .NET Core SDK. I showed you how to create a simple project and briefly explained how the endpoint, the view, and the URL routing system work together. In the next chapter, I show you how to create a simple data-entry application.